Tutoriat 5 (Operating Systems)
Synchronization Examples

Contents

1 Bounded-buffer problem

Recall that the **bounded-buffer problem** (also known as the **producer-consumer** problem) describes a scenario where two or more threads/processes, known as the **producer process** and the **consumer process**, share a common data structure called the **buffer** where the producer generates data and stores them on the buffer, while the consumer takes data off the buffer and consumes it. The two processes share the following variables:

```
#define BUFFER_SIZE 10

typedef struct {
    ...
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int count = 0;
```

The buffer is implemented as a *circular array.* The variable **in** points to the next free position in the buffer, whereas **out** points to the first occupied slot. The variable **count** keeps track of the number of items currently in the buffer. If the buffer is empty, **count == 0**; if the buffer is full, **count == BUFFER_SIZE**.

Producer process

```
item next_produced;

while (true) {
    /* produce an item, store it in
        next_produced */

    while (count == BUFFER_SIZE); /* do
        nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

Consumer process

```
item next_consumed;

while (true) {
    while (count == 0); /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;

    /* consume the item stored in
        next_consumed */
}
```

The issue is that these are two processes that enter their critical sections without following any rules. They execute operations on shared data without a synchronization mechanism. The *count++* and *count−−* instructions are critical and can corrupt the state of the buffer due to race conditions. The following shared data structures can solve this issue:

```
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
```

The binary semaphore *mutex* is used to ensure mutual exclusion. If one of the processes wishes to enter its critical section, it must first acquire the "lock" by calling *wait(mutex)*; once that is all set and done, it can enter its critical section and call *signal(mutex)* upon exiting. The counting semaphore *empty* represents the number of **empty** slots in the buffer. The counting semaphore *full* represents the number of items in the buffer. The new implementation can be found below.

The producer process must first wait for the buffer to have at least one empty slot before storing an item; thus, it calls *wait(empty)*. Once that condition is met, it tries to acquire the mutex. After storing an item, it releases the mutex and increments the number of items in the buffer by calling *signal(full)*. Notice the beginning and ending: it decrements *empty* and increments *full*.

The consumer process must first wait for the buffer to have at least one item before being able to consume; thus, it calls *wait(full)*. Once that condition is met, it tries to acquire the mutex. After consuming an item, it releases the mutex and increments the number of empty slots in the buffer by calling *signal(empty)*. In conclusion, it decrements *full* and increments *empty*.

Producer process

```
item next_produced;

while (true) {
    /* produce an item, store it in
        next_produced */

    wait(empty);
    wait(mutex);

    /* add next_produced to the buffer */

    signal(mutex);
    signal(full);
}
```

Consumer process

```
item next_consumed;

while (true) {
    wait(full);
    wait(mutex);

    /* consume an item from the buffer */

    signal(mutex);
    signal(empty);
}
```

2 Readers-writers problem

The **readers-writers problem** deals with synchronization between multiple threads/processes accesing a shared resource, such as a file or a database. There are two types of processes: **readers** (processes that only **read** the data) and **writers** (processes that both **read** and **modify** the data). While a writer is modifying the data structure, it is necessary to bar other processes from executing, as to prevent them from reading inconsistent data. The proposed synchronization protocol must enforce the following rules:

- **Multiple readers allowed:** any number of readers can be in their critical sections simultaneously.
- **Exclusive writer access:** if a writer is accessing the data, no other process can access it at the same time.

The exclusion pattern here can be called **categorical mutual exclusion**. One process in its critical section does not necessarily exclude other processes, but the presence of one category of processes in the critical section excludes other categories. In terms of priority, there are two versions of this problem:

- **Reader priority:** if a reader is in its critical section, other readers are immediately allowed to enter as well, even if a writer is waiting; the writer must wait for all current readers to finish. **Drawback:** this can lead to **writer starvation**.
- **Writer priority:** once a writer is ready, no new reader may start reading; they must wait until the writer has finished executing in its critical section. **Drawback:** this can lead to **reader starvation.**

The three following variables will be used when designing a solution:

```
int readers = 0;
mutex = Semaphore(1);
empty = Semaphore(1);
```

The **readers** variable keeps track of the amount of readers currently active. The **mutex** variable is a binary semaphore used to increment and decrement **readers**. The **empty** variable is a binary semaphore that is 1 if there are no processes (readers or writers) in their critical sections, and 0 otherwise.

2.1 Reader priority

2.1.1 Writer process

A writer can enter its critical section when there are no other processes executing in their critical sections. The code for the writers would be as follows:

```
wait(empty);
/* critical section (writing operations) */
signal(empty);
```

2.1.2 Reader process

There can be multiple readers executing in their critical sections at the same time. When the first reader wishes to enter its critical section, it must first check with the **empty** semaphore to see if there is already a writer process executing. Once there is no writer active, the reader is allowed to enter its critical section. It uses the **mutex** semaphore to safely increment the variable **readers**. Once the first reader has successfully made its appearance, subsequent readers do not have to check with the **empty** semaphore; as long as there is one reader active, there is no writer, so they only have to increment the variable **readers**.

When a reader exits its critical section, it has to make use of the **mutex** semaphore to decrement the variable **readers**. When the last reader exits its critical section, it has the additional responsability of updating the **empty** semaphore, signaling that there are no processes left executing in their critical sections, giving the green light for any waiting writer.

```
/* 1. increment readers and check if it's the FIRST reader */
wait(mutex);
readers++;
if (readers == 1) {
    wait(empty);
}
signal(mutex);

/* 2. reading operations */

/* 3. decrement readers and check if it's the LAST reader */
wait(mutex);
readers--;
if (readers == 0) {
    signal(empty);
}
signal(mutex);
```

This solution can lead to **writer starvation**, since reader processes can always keep entering their critical sections, leaving writer processes to wait indefinitely.

2.2 Writer priority

To ensure that writer processes have priority over the readers, some additional variables shall be introduced. The shared data becomes:

```
int readers = 0;
int writers = 0;

empty = Semaphore(1);
rmutex = Semaphore(1);
wmutex = Semaphore(1);
tryRead = semaphore(1);
```

The **writers** variable keeps track of the amount of writers that wish to start writing. The semaphore **rmutex** is used for incrementing the **readers** variable, and **wmutex** for incrementing the **writers** variable. A writer process can only start executing when there are no other active processes, which is what the **empty** semaphore signals. The semaphore **tryRead** blocks additional readers from entering if there's a writer process waiting.

2.2.1 Reader process

If there are no writers waiting, the reader process can do the whole *increment "readers" - start reading - decrement "readers"* cycle, while also checking if it's the first and the last reader.

```
/* 1. try to gain reading permission (writers can block additional readers from entering) */
wait(tryRead);

/* 2. increment the readers variable and check if it's the first reader  */
wait(rmutex);
readers++;
if (readers == 1) {
    wait(empty);
}
signal(rmutex);
signal(tryRead);

/* 3. perform reading operations */

/* 4. decrement the readers variable and check if it's the last reader to exit */
wait(rmutex);
readers--;
if (readers == 0) {
    signal(empty);
}
signal(rmutex);
```

2.2.2 Writer process

When the first writer process arrives, it must block additional readers from starting to read. When the last writer process leaves, it must unblock the readers. Other than that, each writer performs the *increment "writers" - perform writing - decrement "writers"* cycle.

```
/* 1. increment the number of writers waiting to start writing; if it's the first writer, block
    additional readers from starting to read */
wait(wmutex);
writers++;
if (writers == 1) {
    wait(tryRead);
}
signal(wmutex);

/* 2. perform writing operations */
wait(empty);
/* write */
signal(empty);

/* 3. decrement the number of writers; if it's the last writer, unblock the readers */
wait(wmutex);
writers--;
if (writers == 0) {
    signal(tryRead);
}
signal(wmutex);
```

3 Dining-philosophers problem

Imagine a number of philosophers sitting around at a circular dining table. Each philosopher's life consists of an alternating cycle between **thinking** and **eating**. Each philosopher has a pair of two shared forks (or utensils): a philosopher shares his right-side fork with the philosopher on his right (who sees it as his left-side fork); the left-side fork is shared with the philosopher on his left (who sees it as his right-side fork).

To eat, a philosopher must pick up **both** utensils; after they're done eating, the utensils are placed back in their initial positions and the philosopher goes back to thinking. It is possible that a philosopher will only be able to pick up one utensil as the other could already be in use by an adjacent philosopher; in this case, the philosopher will be kept waiting until the other utensil becomes available.

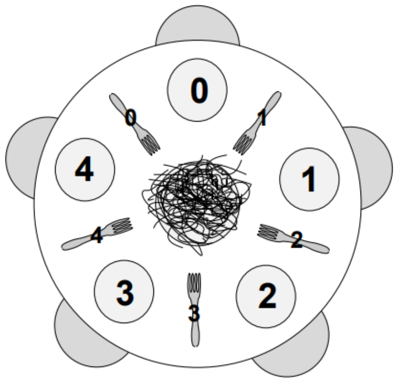

Figure 1: Five dining philosophers

The issue is the possibility of a **deadlock**. If every single philosopher picked up the utensil on their left, there would be no more utensils available, and all philosophers would be kept waiting indefinitely for the utensil on the right.

The **philosophers** can be viewed as being **threads/processes**, and the forks are **shared data**. The goal is to ensure a deadlock-free environment, where philosophers can safely alternate between thinking and eating without waiting indefinitely for resources.

3.1 Semaphore solution (deadlock issue)

```
while (true) {
    // pick up chopsticks (left and then right)
    wait(chopsticks[i]);
    wait(chopsticks[i + 1] % chopstick_count);

    eat();

    // put down chopsticks (left and then right)
    signal(chopsticks[i]);
    signal(chopsticks[i + 1] % chopstick_count);

    think();
}
```

This solution ensures **mutual exclusion**. However, if all philosophers pick up the left-side utensil at the same time, a **deadlock** takes place. There are a number of ways to fix this solution:

1. **Resource hierarchy:** ensure that the lower index chopstick is always picked up first. Take a look at the five-philosopher diagram: both philosophers 0 and 4 will try to pick up the utensil 0, and only one of them will succeed. If 4 succeeds, then the utensil 1 will remain available; if 0 succeeds, then the utensil 4 will remain available. By looking at the diagram and trying to visualize different execution paths for the remaining philosophers, it can be seen that no deadlocks will take place.

2. **Asymmetric allocation:** different pick-up strategies can be made up, based on the philosopher's index. Philosophers with odd indexes first pick up the left utensil, and then the one on the right. Philosophers with

even indexes pick up their utensils the other way around.

3. **Arbitrator solution:** either both chopsticks are picked up at the same time, or neither is picked up. This can be done using a mutex lock.

3.2 Monitor solution

Instead of having the philosophers fight over chopsticks, they interact with a **monitor**. Each philosopher can be in one of three states: **thinking**, **hungry**, or **eating**.

When a philosopher wants to eat, they set their state to **hungry**. A philosopher can only go from **hungry** to **eating** if their **left** and **right** neighbors are not eating.

Once a philosopher has finished eating, they set their state back to **thinking**. If a neighbor was waiting (sleeping) because this specific philosopher was busy eating, then the neighbor is woken up.

```
monitor DiningPhilosophers
{
    enum {THINKING; HUNGRY, EATING};
    state[N];
    self[N];

    // wants to start eating
    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);

        if (state[i] != EATING) {
            self[i].wait;
        }
    }

    // finished eating
    void putdown(int i) {
        state[i] = THINKING;

        // signal left and right neighbors
        test((i + 1) % N);
        test((i - 1) % N);
    }

    // checks if it is safe to start eating
    void test(int i) {
        if ((state[i] == HUNGRY) && (state[(i + 1) % N] != EATING) && (state[(i - 1) % N] !=
            EATING)) {
            state[i] = EATING;
            self[i].signal();
        }
    }

    initialization_code() {
        for (int i = 0; i < N; i++)
            state[i] = THINKING;
    }
}
```

```
while (true) {
    DiningPhilosophers.pickup(i);

    eat();

    DiningPhilosophers.putdown(i)

    think();
}
```

4 Barbershop problem

(TODO inca nu sunt terminate explicatiile)
Imagine a barbershop with **one barber** who has **one barber chair**. There is also a **waiting room** with **N chairs** for **customers**. If there are no customers, the barber falls asleep in his chair. When he finishes a haircut, he checks the waiting room; if it's empty, he goes back to sleep in his chair.

If a customer arrives and the barber is asleep, the customer wakes him up to get a haircut. If the barber is busy but there are free chairs in the waiting room, the customer sits down and waits; however, if there are no waiting chairs available, the customer leaves.

4.1 Semaphore solution

The **barber process** and the **customer processes** will share the following variables:

```
ccmutex = Semaphore(1);

customer = Semaphore
barber = Semaphore(1);

barberDone = Semaphore(1);
customerDone = Semaphore(1);
```

4.1.1 Barber process

```
while (true) {
    /* 1. wait for a customer to arrive ("wakeup handshake") */
    wait(customer);
    signal(barber);

    cutHair();

    /* 2. signal that the haircut is done ("goodbye handshake") */
    signal(barberDone);
    wait(customerDone);
}
```

4.1.2 Customer process

```
while (true) {
    /* 1. if the shop is full, leave */
    wait(ccmutex);
    if (customerCount == maxCustomerCount) {
        signal(ccmutex);
        continue;
    }

    /* 2. increment the number of customers */
    customerCount++;
    signal(ccmutex);

    /* 3. wake the barber up ("wakeup handshake") */
    signal(customer);
    wait(barber);

    getHaircut();

    /* 4. haircut is finished ("goodbye handshake") */
    signal(customerDone);
    wait(barberDone);

    /* 5. decrement the number of customers */
    wait(ccmutex);
    customerCount--;
    signal(ccmutex);

    doStuffUntilNextHaircut();
}
```

The issue with this solution is that the customers are not necessarily picked in the order of their arrival, so a customer can be wait indefinitely.

4.2 Queue-based solution